IO

vs. NIO

micro-23

Aliaksei Bialiauski

Designed in LaTeX

Thread Management

Synchronous communication

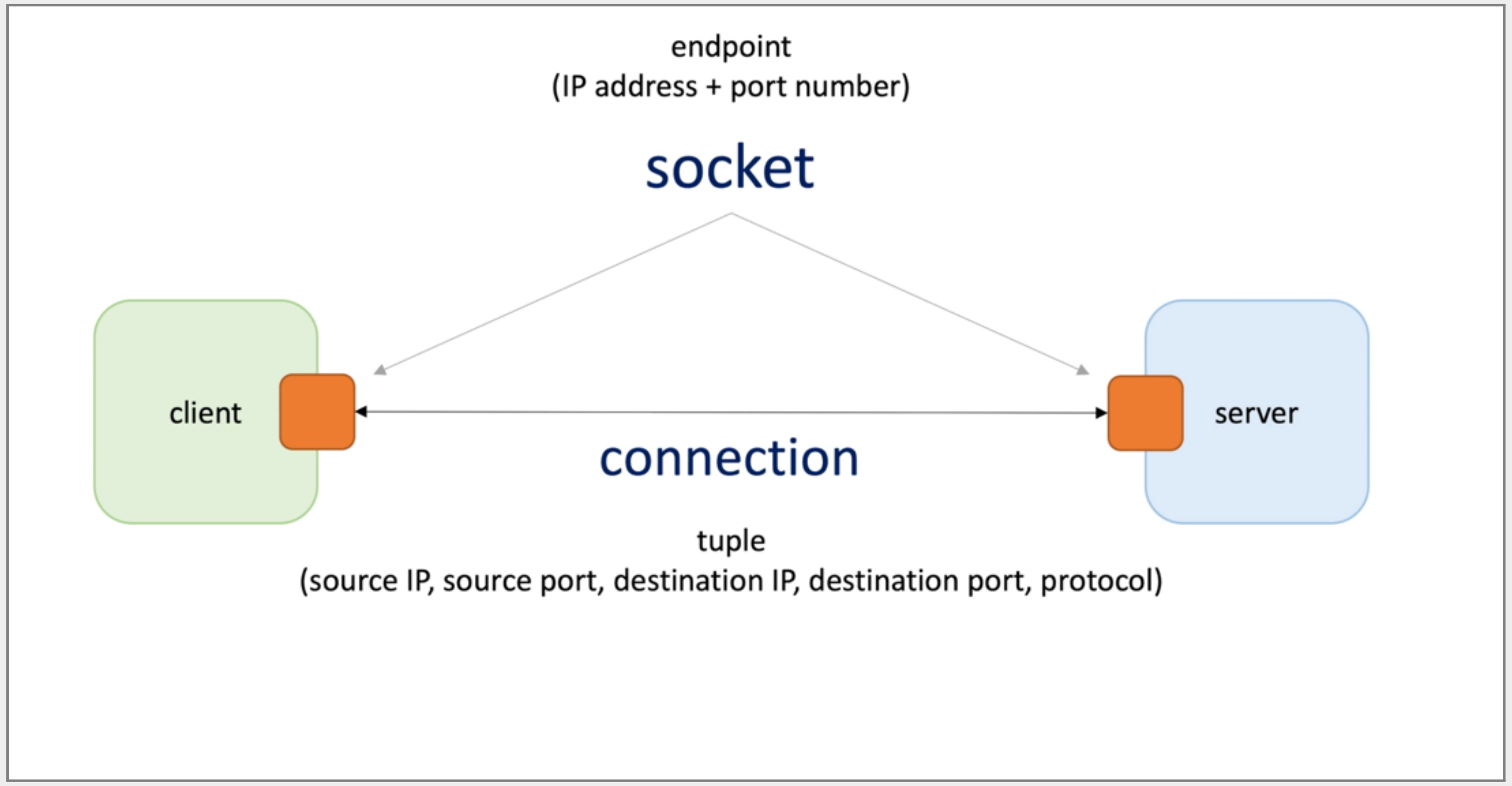
Reactive Streams

Project Reactor

Chapter #1:

Thread Management

Socket


endpoint
(IP address + port number)
socket
client
server
connection
tuple
(source IP, source port, destination IP, destination port, protocol)

blocking socket vs. non-blocking socket

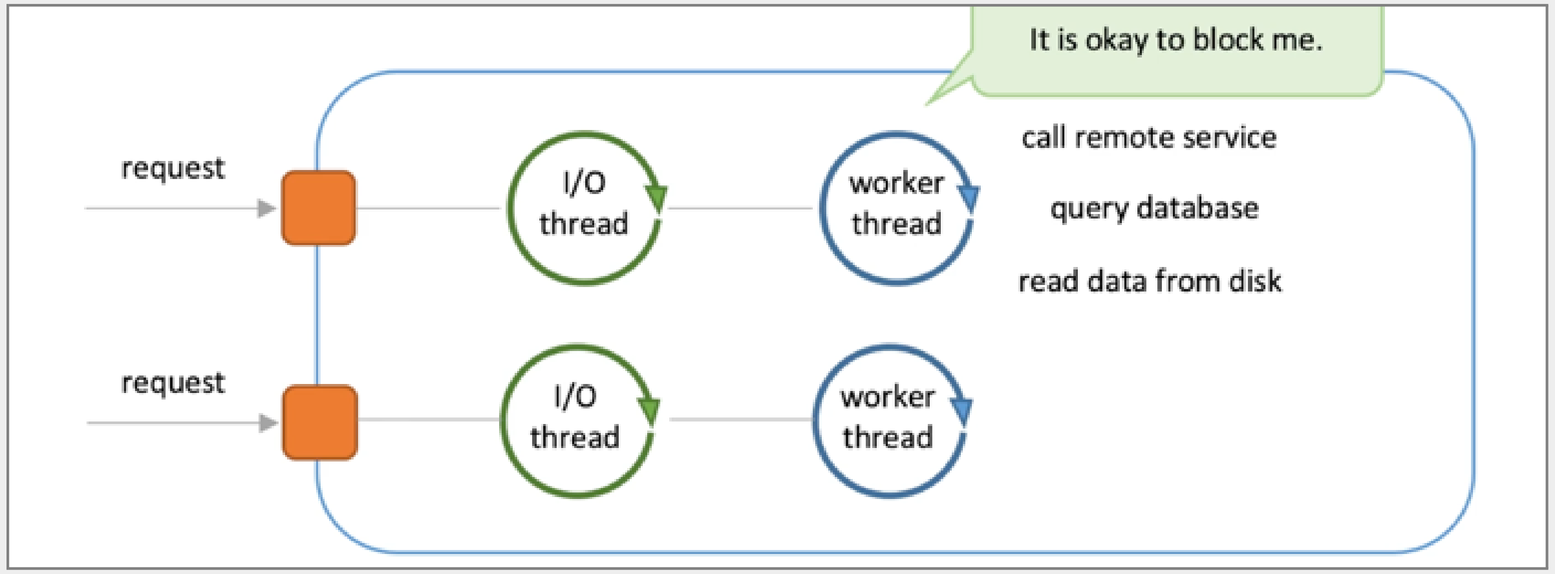
blocking

thread is suspended until read/write from/to socket completes

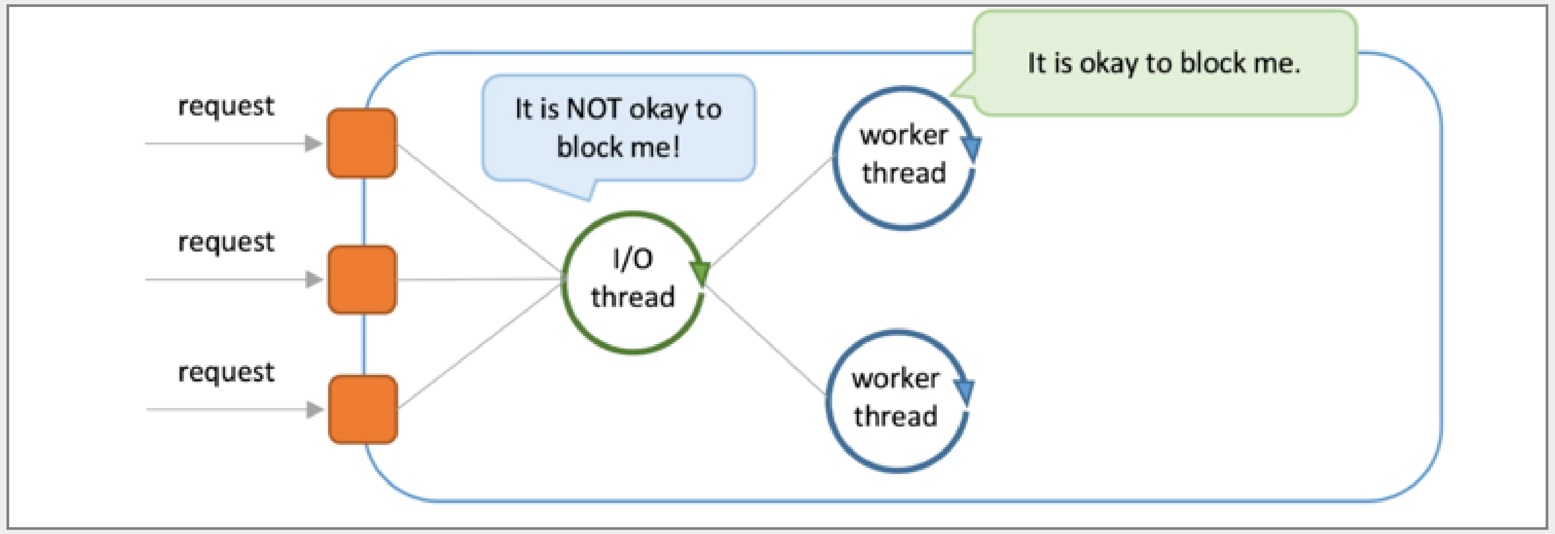
non-blocking

thread reads data available in the socket buffer and does not wait for the remaining data to arrive

Thread per connection

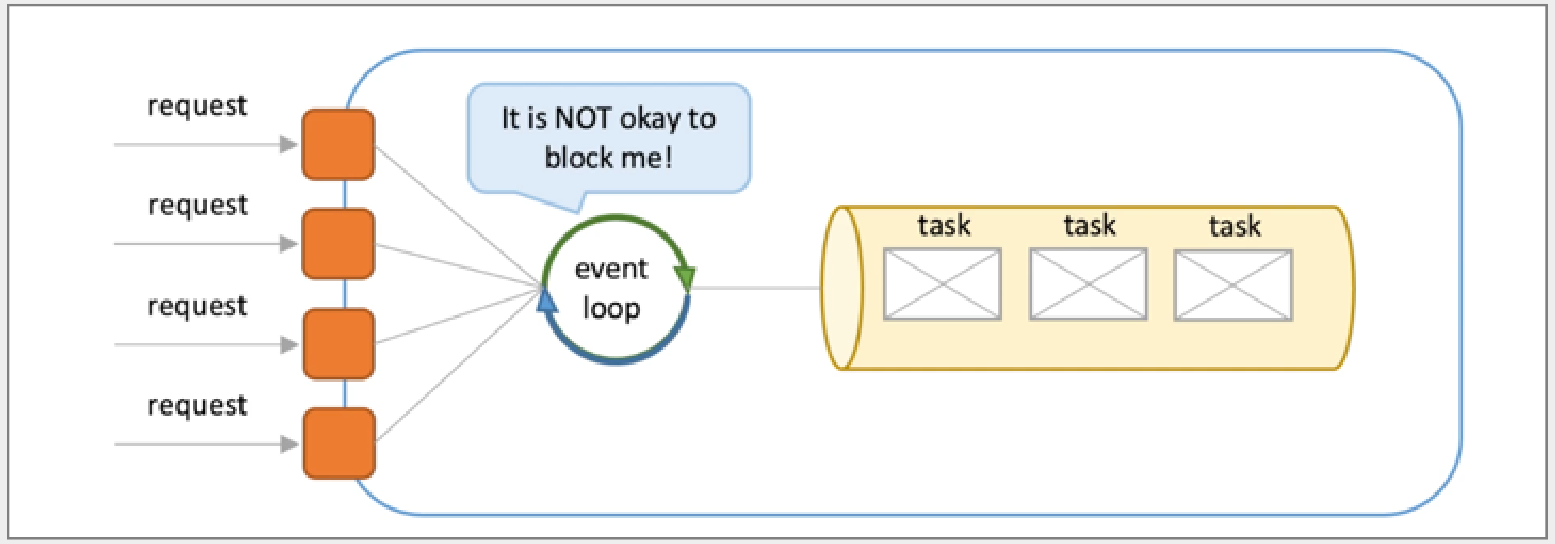


Thread per request



https://tomcat.apache.org

Event Loop



https://netty.io

Concurrency vs. Parallelism

At any given time 1 CPU can run only thread. Parallel execution happens when we have multiple CPUs.

IO vs. CPU bounds

Event loop is good for IO-bound workload (e.g. large files).

Thread per request is good for CPU-bound workload.

Chapter #2:

Synchronous communication

REST communication

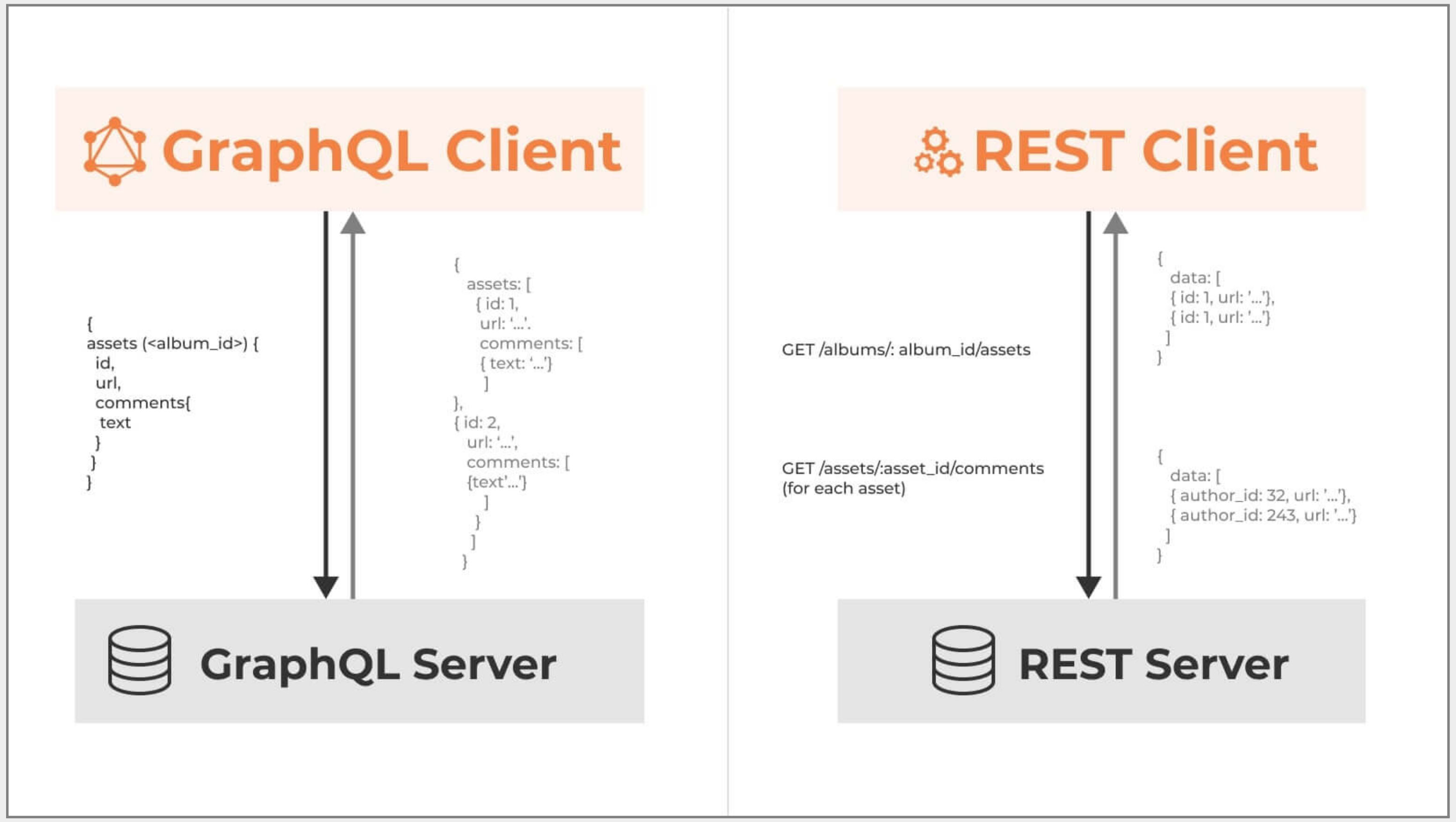
```
class RestExample {

  private final RestTemplate rest;

  public ResponseEntity<String> fetch(final String url) {
    return rest.getForEntity(url + "/1", String.class);
  }

  public static void main(String[] args) {
    final ResponseEntity<String> entity = new RestExample(
      new RestTemplate()
    ).fetch("http://localhost:8080/spring-rest/foos");
  }
}
```

Open Feign - Declarative REST Client

```
@FeignClient(name = "stores", url = "${api.url}", configuration = FooConfiguration.class)
public interface StoreClient {
  @GetMapping(value = "/stores", produces = "application/xml")
  List<Store> stores();

  @GetMapping("/stores")
  Page<Store> stores(Pageable pageable);

  @PutMapping(url = "/stores/{storeId}", consumes = "application/json")
  Store update(@PathVariable Long storeId, Store store);

  @DeleteMapping(value = "/stores/{storeId:\\d+}")
  void delete(@PathVariable Long storeId);
}
```

https://docs.spring.io/spring-cloud-openfeign/docs/current/reference/html/

GraphQL



```
{
assets (<album_id>) {
 id,
 url,
 comments{
  text
 }
 }
}
```

```
{
  assets: [
   { id: 1,
    url: '...'.
    comments: [
    { text: '...'}
     ]
 },
 { id: 2,
  url: '...',
  comments: [
  {text'...'}
     ]
    }
   ]
  }
```

```
GET /albums/: album_id/assets
```

```
{
  data: [
  { id: 1, url: '...'},
  { id: 1, url: '...'}
 ]
}
```

```
GET /assets/:asset_id/comments
(for each asset)
```

```
{
  data: [
  { author_id: 32, url: '...'},
  { author_id: 243, url: '...'}
 ]
}
```

GraphQL Schema

```
type Post {
    id: ID!
    title: String!
    text: String!
    category: String
    author: Author!
}

type Author {
    id: ID!
    name: String!
    thumbnail: String
    posts: [Post]!
}

type Query {
    recentPosts(count: Int, offset: Int): [Post]!
}

type Mutation {
    createPost(title: String!, text: String!, category: String, authorId: String!) : Post!
}
```

GraphQL Query

```
query {
    recentPosts(count: 10, offset: 0) {
        id
        title
        category
        author {
            id
            name
            thumbnail
        }
    }
}
```

GraphQL Mutation

```
mutation {
    createPost(authorId: "256", text: "book.tex", title: "Code Ahead") {
        id,
        title
    }
}
```

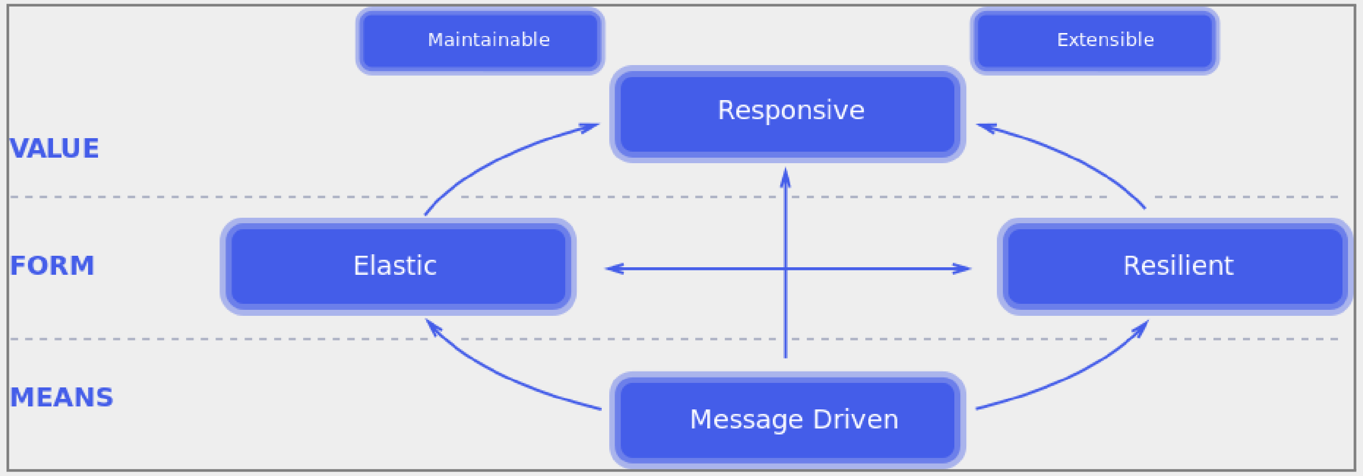
Spring Controllers

```
@Controller
public class PostController {

  private Posts posts;

  @QueryMapping
  public List<Post> recentPosts(@Argument int count, @Argument int offset) {
    return posts.recent(count, offset);
  }
}
```

https://graphql.org/learn/

Chapter #3: Reactive Streams

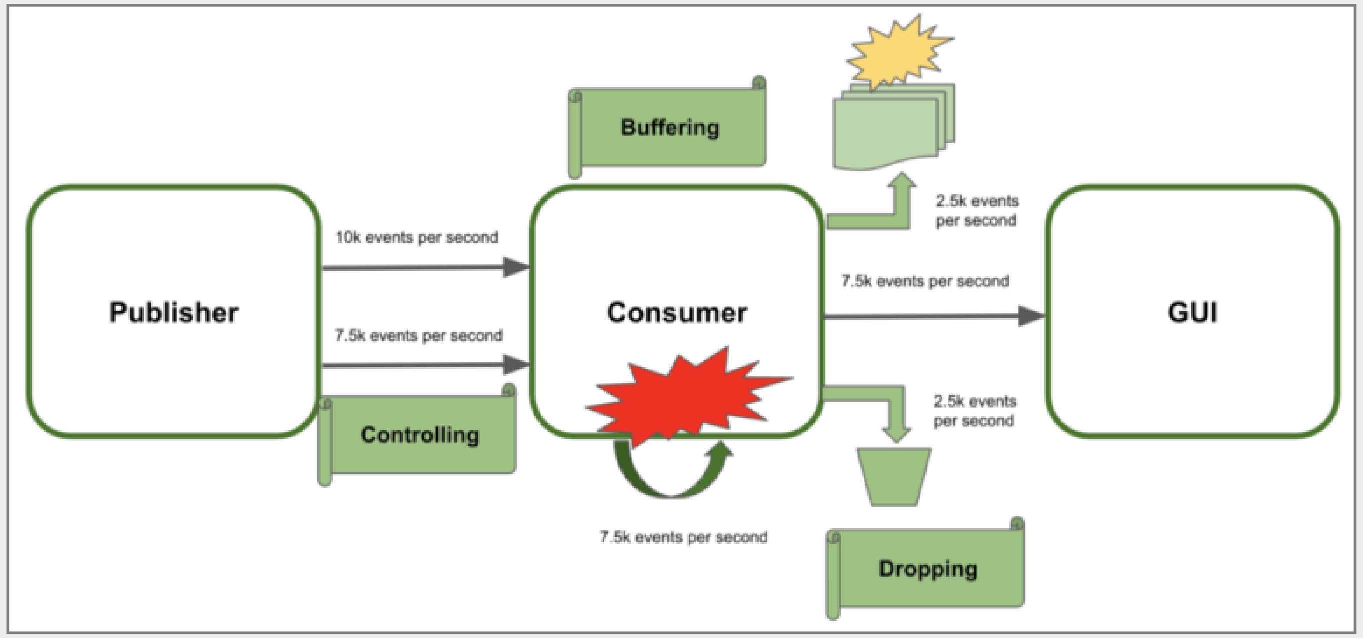
Reactive Manifesto



https://www.reactivemanifesto.org

Backpressure

In Reactive Streams, backpressure also defines how to regulate the transmission of stream elements.



Reactive Streams

"Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols."

— Reactive Streams website

Publisher

```
public interface Publisher<T> {
  public void subscribe(Subscriber<? super T> s);
}
```

Subscriber

```
public interface Subscriber<T> {
  public void onSubscribe(Subscription s);
  public void onNext(T t);
  public void onError(Throwable t);
  public void onComplete();
}
```

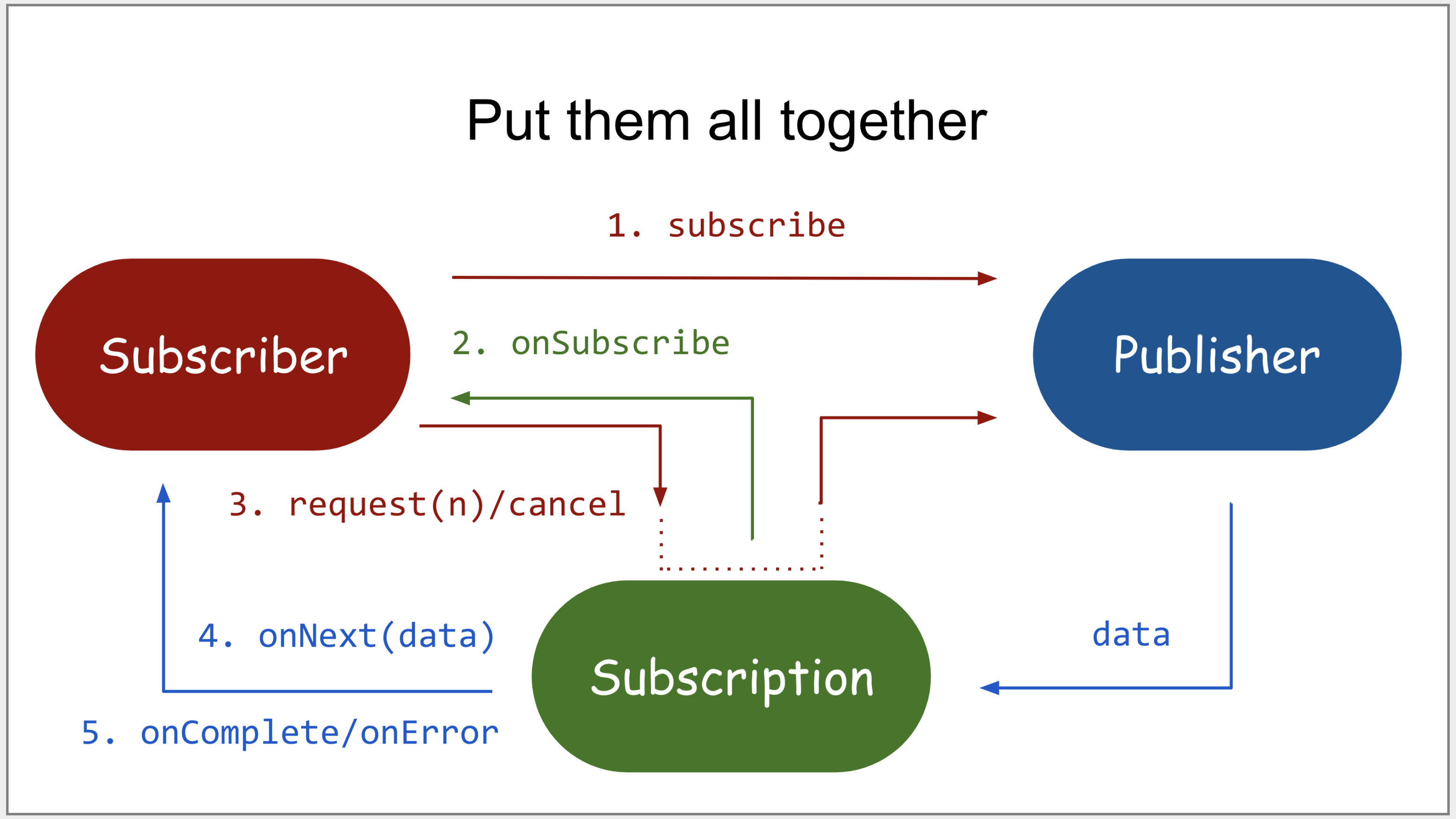
Subscription

```
public interface Subscription {
  public void request(long n);
  public void cancel();
}
```

Processor

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

Pub-Sub Flow


Put them all together
1. subscribe
Subscriber
2. onSubscribe
Publisher
3. request(n)/cancel
4. onNext(data)
Subscription
data
5. onComplete/onError

Implementations

https://github.com/ReactiveX/RxJava

https://github.com/eclipse-vertx/vert.x

https://github.com/smallrye/smallrye-mutiny

https://github.com/reactor/reactor-core

Chapter #4: Project Reactor

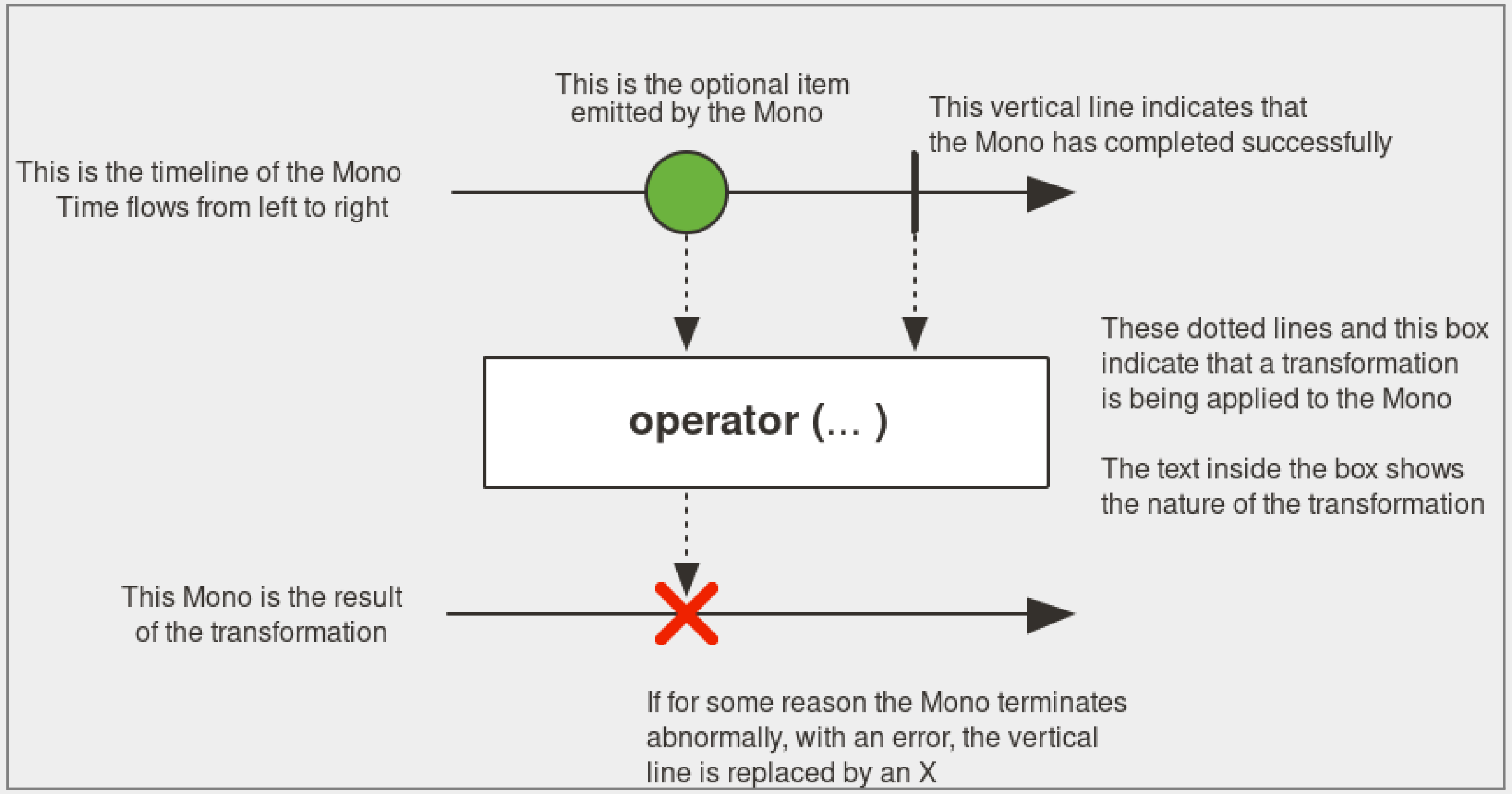
Project Reactor

“Well-suited for a microservices architecture, Reactor offers backpressure-ready network engines for HTTP (including Websockets), TCP, and UDP.”

“Reactor offers two reactive and composable APIs, contains 2 publishers: Flux [N] and Mono [0|1], which extensively implement Reactive Extensions.”

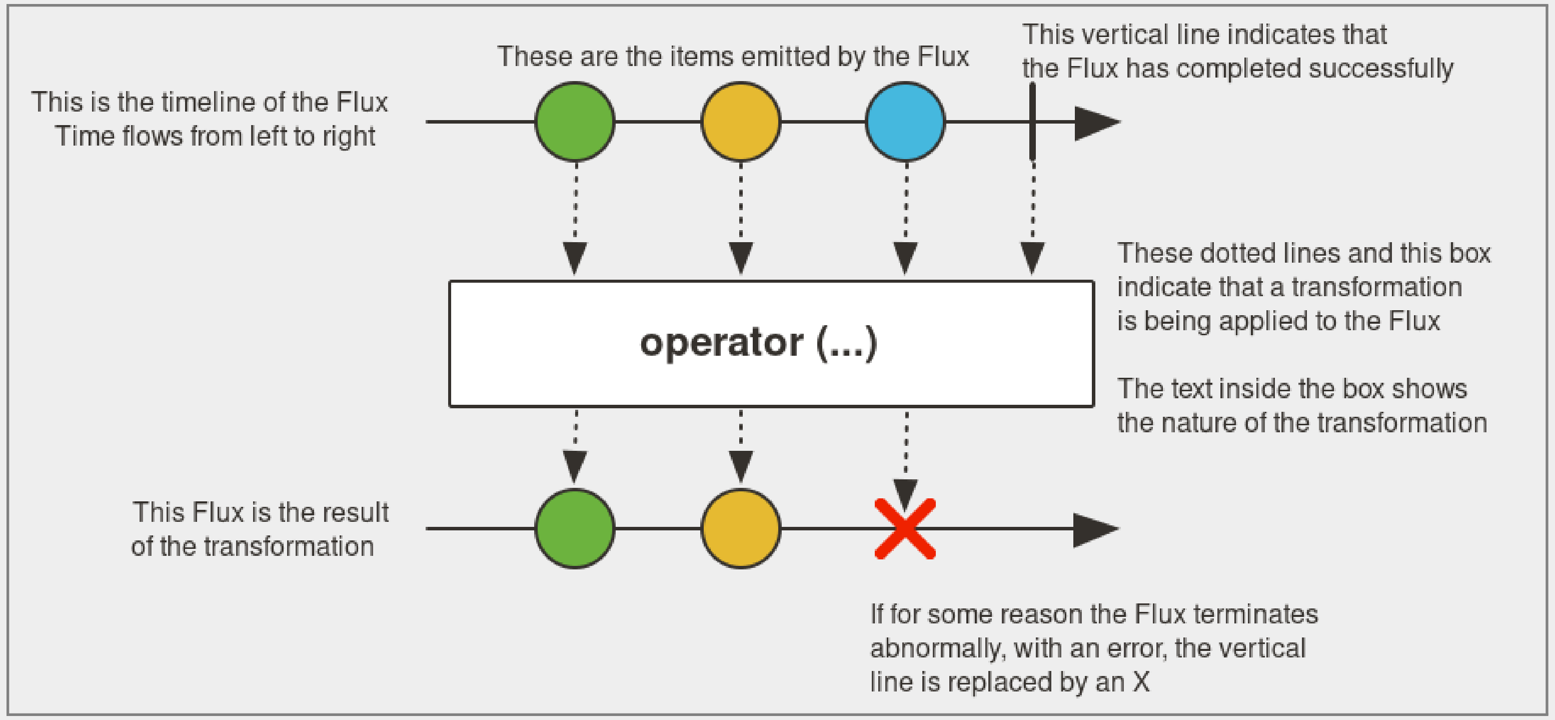
— Reactor website

Mono, publisher with asynchronous 0–1 result.


This is the optional item emitted by the Mono
This vertical line indicates that the Mono has completed successfully
This is the timeline of the Mono Time flows from left to right
These dotted lines and this box indicate that a transformation is being applied to the Mono
The text inside the box shows the nature of the transformation
operator (...)
This Mono is the result of the transformation
If for some reason the Mono terminates abnormally, with an error, the vertical line is replaced by an X

```
class MonoExample {
  public static void main(String[] args) {
    Mono<String> noData = Mono.empty();
    Mono<String> data = Mono.just("foo");

  }
}
```

Flux, publisher with asynchronous sequence of 0–N items.


These are the items emitted by the Flux
This vertical line indicates that the Flux has completed successfully
This is the timeline of the Flux Time flows from left to right
These dotted lines and this box indicate that a transformation is being applied to the Flux
operator (...)
The text inside the box shows the nature of the transformation
This Flux is the result of the transformation
If for some reason the Flux terminates abnormally, with an error, the vertical line is replaced by an X

```java
class FluxExample {
  public static void main(String[] args) {
    Flux<String> seq1 = Flux.just("foo", "bar", "foobar");
    List<String> iterable = Arrays.asList("foo", "bar", "foobar");
    Flux<String> seq2 = Flux.fromIterable(iterable);
    Flux<Integer> range = Flux.range(1, 25);
  }
}
```

Immutability

```
class ImmutabilityExample {
  void count() {
    Flux<String> count = Flux
      .fromStream(this.returnStream())
      .take(10)
      .flatMap(
        c -> Flux.zip(Mono.just(c), Mono.fromCompletionStage(this.returnCompletableFuture(c)))
      )
      .map(tuple -> tuple.getT2() + " #" + tuple.getT1()); // count is not there
    count.subscribe(System.out::println); // count is running
  }
}
```

Declarativity

Other reactor projects

https://github.com/reactor/reactor-kafka

https://github.com/reactor/reactor-netty

Spring Reactive

Spring supports reactive programming using Project Reactor.

Servlet Web -> Reactive Web

JDBC -> R2DBC

Imperative -> Declarative

```
interface UserRepository extends ReactiveRepository {

  Mono<User> findById(Long id);

  Flux<User> findAllByFirstName(String firstname);
}


class Users {

  private final UserRepository repository;

  public Mono<User> user(Long id) {
    return this.repository.findById(id);
  }

  public Mono<User> transform() {
    return this.repository.findById(1L)
      .flatMap(
        user -> {
          user.setInfo(new UserInfo("Palo Alto/CA", "Safari", "h1alexbel/transformed"));
          return Mono.just(user);
        }
      );
  }
}
```

```
@Configuration
@RequiredArgsConstructor
class UserRoutes {

  private final Users user;

  @Bean
  public RouterFunction<ServerResponse> user() {
    return RouterFunctions.route()
      .GET("/api/v1/users/{id}",
        req ->
          ServerResponse.ok().body(
            this.users.findById(req.pathVariable("id")), User.class
          )
      ).build();
  }
}
```